[OpenGL 3D 2018 第11回]

ゾンビアクターサガ

# アクタークラス

## Actor.hを作成する

人間モデルが動くだけではゲームにならないので、何か要素を付け足さなければなりません。ありきたりなアイデアですが、「次々にわいてくるゾンビを撃って倒すゲーム」にしようと思います。  
ゲーム中には、プレイヤー、弾、ゾンビ、それに家や木などの障害物が登場します。つまり、これらの様々な物体を表現する機能を作る必要があるわけです。そこで、Actor(あくたー)というクラスを作ることにしましょう。

Srcフォルダに「Actor.h」という名前のヘッダーファイルを追加してください。そして追加したファイルを開き、以下のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Actor.h  
**+**\*/  
**+**#ifndef ACTOR\_H\_INCLUDED  
**+**#define ACTOR\_H\_INCLUDED  
**+**#include <GL/glew.h>  
**+**#include "Shader.h"  
**+**#include "MeshList.h"  
**+**#include <glm/vec3.hpp>  
**+**#include <vector>  
**+**  
**+**/\*\*  
**+**\* 直方体.  
**+**\*/  
**+**struct Rect  
**+**{  
**+** glm::vec3 origin;  
**+** glm::vec3 size;  
**+**};  
**+**  
**+**/\*\*  
**+**\* シーンに配置するオブジェクト.  
**+**\*/  
**+**class Actor  
**+**{  
**+**public:  
**+** Actor() = default;  
**+** virtual ~Actor() = default;  
**+**  
**+** void Initialize(int mesh, GLuint tex, int hp,  
**+** const glm::vec3& pos, const glm::vec3& rot, const glm::vec3& scale);  
**+** void Finalize();  
**+** virtual void Update(float deltTIme);  
**+**  
**+**public:  
**+** int mesh = 0;  
**+** GLuint texture = 0;  
**+**  
**+** glm::vec3 position;  
**+** glm::vec3 rotation;  
**+** glm::vec3 scale;  
**+**  
**+** glm::vec3 velocity;  
**+** int health = 0;  
**+** Rect colLocal;  
**+** Rect colWorld;  
**+**};  
**+**  
**+**Actor\* FindAvailableActor(std::vector<Actor\*>&);  
**+**void UpdateActorList(std::vector<Actor\*>&, float);  
**+**void RenderActorList(std::vector<Actor\*>&, Shader::Program&, MeshList&);  
**+**void ClearActorList(std::vector<Actor\*>&);  
**+**  
**+**#endif // ACTOR\_H\_INCLUDED

今回作成するActorクラスは、派生クラスを作って使うことを想定しています。そのため、デストラクタとUpdate関数にvirtual(ばーちゃる)を指定しておきました。FindAvailableActor(ふぁいんど・あべいらぶる・あくたー)関数などの4つの関数は、複数のActorを処理するための関数たちです。  
今回は、引数としてActor配列の範囲を指定するのではなく、std::vector<Actor\*>型の参照を指定するようにしました。配列の範囲の先頭と終端を指定するのと比べると、引数の数が1つ少なくなるので、その分だけプログラムを書く手間を減らせます。欠点はstd::vectorしか使えなくなることです。しかしこれは、大した問題にはならないと思います。配列よりvectorのほうが便利なことが多いわけですから。

## Actor::Initialize関数を定義する

続いてActorクラスのメンバ関数を定義していきます。上から順番にということで、Initialize(いにしゃらいず)関数から始めましょう。SrcフォルダにActor.cppというファイルを追加してください。そして追加したActor.cppを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Actor.cpp  
**+**\*/  
**+**#include "Actor.h"  
**+**#include <algorithm>  
**+**#include <glm/gtc/matrix\_transform.hpp>  
**+**#include <glm/gtc/quaternion.hpp>  
**+**  
**+**/\*\*  
**+**\* アクターを初期化する.  
**+**\*  
**+**\* @param mesh アクターとして表示するメッシュID.  
**+**\* @param tex メッシュに貼るテクスチャのID.  
**+**\* @param hp アクターの耐久力.  
**+**\* @param pos メッシュを表示する座標.  
**+**\* @param rot メッシュを表示する回転角度.  
**+**\* @param scale メッシュを表示する大きさ.  
**+**\*/  
**+**void Actor::Initialize(int mesh, GLuint tex, int hp,  
**+** const glm::vec3& pos, const glm::vec3& rot, const glm::vec3& scale)  
**+**{  
**+** this->mesh = mesh;  
**+** texture = tex;  
**+** position = pos;  
**+** rotation = rot;  
**+** this->scale = scale;  
**+** health = hp;  
**+**}

Initialize関数はアクターの初期設定を行う関数です。アクターにはさまざまなパラメーターがあるので、それらを設定するために多くの引数を受け取らなければなりません。  
ところで、このプログラムには「this->」を使っている部分があります。C++では引数とメンバ変数に同じ名前をつけることが許されています。そして、その名前だけを書いた場合は「引数」とみなされることになっています。しかし、そのままではメンバ変数を扱えないので、名前の前に「this->」をつけるとメンバ変数扱いとなるようになっているのです。例えば上記のプログラムの場合、「mesh」と書くと引数のmeshになり、「this->mesh」と書くとメンバ変数のmeshになるわけです。

なお、すべての引数で同じ名前になっていない理由は単純です。「本当は引数とメンバ変数にはすべて違う名前を付けたかったけれど、meshやscaleは単語の文字数が少なすぎて、短縮名を思いつかなかったから」です。

## Actor::Finalize関数を定義する

続いてFinalize(ふぁいならいず)関数を定義しましょう。Initialize関数の定義の下に、次のプログラムを追加してください。

this->scale = scale;  
 health = hp;  
 }  
**+**  
**+**/\*\*  
**+**\* アクターを破棄する.  
**+**\*/  
**+**void Actor::Finalize()  
**+**{  
**+** health = 0;  
**+**}

Finalize関数ではhealthを0にしているだけです。

## Actor::Update関数を定義する

次はUpdate(あっぷでーと)関数を定義します。Finalize関数の定義の下に、次のプログラムを追加してください。

void Actor::Finalize()  
 {  
 }  
**+**  
**+**/\*\*  
**+**\* アクターを更新する.  
**+**\*  
**+**\* @param deltaTime 経過時間.  
**+**\*/  
**+**void Actor::Update(float deltaTime)  
**+**{  
**+** position += velocity \* deltaTime;  
**+** colWorld.origin = colLocal.origin + position;  
**+** colWorld.size = colLocal.size;  
**+**}

Update関数ではアクターの座標(position)と衝突判定の位置を更新します。  
ここで衝突判定用の直方体を計算している理由は、衝突判定を行うたびに衝突判定用の直方体をワールド空間に持ってくるのは、計算時間がもったいないからです。更新関数でワールド空間における直方体の座標を計算しておくことで、実際に衝突判定を行うときには、計算後の直方体同士で判定を行えるようにしているわけです。

## std::vector<Actor\*>から未使用のアクターを見つける

ここからは、Actor\*(あくたー・ぽいんた)型のstd::vector(えすてぃーでぃー・べくたー)、つまりstd::vector<Actor\*>(あくたー・ぽいんた・べくたー)型に使う関数を定義していきます。

今回は理由があってstd::vector<Actor\*>を初期化する関数を定義していません。ですので、まずは未使用のアクターを見つけるFindAvailableActor(ふぁいんど・あべいらぶる・あくたー)関数から定義を始めましょう。Actor::Update関数の定義の下に、次のプログラムを追加してください。

colWorld.origin = colLocal.origin + position;  
 colWorld.size = colLocal.size;  
}  
**+**  
**+**/\*\*  
**+**\* 利用可能なアクターを取得する.  
**+**\*  
**+**\* @param actorList 検索対象のアクターのリスト.  
**+**\*  
**+**\* @return 利用可能なアクターのポインタ.  
**+**\* 利用可能なアクターが見つからなければnullptr.  
**+**\*/  
**+**Actor\* FindAvailableActor(std::vector<Actor\*>& actorList)  
**+**{  
**+** for (auto& actor : actorList) {  
**+** if (actor && actor->health <= 0) {  
**+** return actor;  
**+** }  
**+** }  
**+** return nullptr;  
**+**}

この関数が扱うのはActorそのものではなくActorのポインタです。そのため、healthを確認する前に、Actorポインタがnullptrでないことを確認しなければなりません。これを行っているのが「if (actor &&」の部分です。ポインタ型の変数をif文の条件式として使うと、変数がnullptrならばfalse、そうでなければtrueとして評価されます。また、「&&(論理積)」は常に「左辺→右辺」の順番で実行されます。このとき、左辺の式がfalseの場合は右辺の式は実行されません。  
上記のif文では、まず左辺の式でActorポインタがnullptrかどうかがチェックされます。nullptrだった場合はfalseとなり、右辺の「actor->health <= 0」は実行されません。nullptrでなかった場合はtrueとなるので「actor->health <= 0」が実行されます。

## std::vector<Actor\*>に含まれるアクターを更新する

続いてstd::vector<Actor\*>に含まれるアクターの状態を更新する関数を定義します。  
FindAvailableActor関数の定義の下に、次のプログラムを追加してください。

if (actor && actor->health <= 0) {  
 return actor;  
 }  
 }  
 return nullptr;  
 }  
**+**  
**+**/\*\*  
**+**\* アクターの状態を更新する.  
**+**\*  
**+**\* @param actorList 更新するアクターのリスト.  
**+**\* @param deltaTime 前回の更新からの経過時間.  
**+**\*/  
**+**void UpdateActorList(std::vector<Actor\*>& actorList, float deltaTime)  
**+**{  
**+** for (auto& actor : actorList) {  
**+** if (actor && actor->health > 0) {  
**+** actor->Update(deltaTime);  
**+** }  
**+** }  
**+**}

FindAvailableActor関数と同様に、Actorポインタがnullptrではないことを確認してからhealthを調べるようにしています。nullptrではなくhealthも1以上ならActor::Update関数を呼び出します。

## std::vector<Actor\*>に含まれるアクターを描画する

次に、std::vector<Actor\*>に含まれるアクターを描画する関数を作ります。  
UpdateActorList関数の定義の下に、次のプログラムを追加してください。

if (actor && actor->health > 0) {  
 actor->Update(deltaTime);  
 }  
 }  
}  
**+**  
**+**/\*\*  
**+**\* アクターを描画する.  
**+**\*  
**+**\* @param actorList 描画するアクターのリスト.  
**+**\* @param shader 描画に使用するシェーダー・オブジェクト.  
**+**\* @param meshList 描画に使用するメッシュリスト.  
**+**\*  
**+**\* 事前にShader::Use(), Shader::SetViewProjectionMatrix()等の関数を実行しておくこと.  
**+**\*/  
**+**void RenderActorList(std::vector<Actor\*>& actorList,  
**+** Shader::Program& shader, MeshList& meshList)  
**+**{  
**+** for (auto& actor : actorList) {  
**+** if (actor && actor->health > 0) {  
**+** shader.BindTexture(0, actor->texture);  
**+** shader.Draw(meshList.Get(actor->mesh),  
**+** actor->position, actor->rotation, actor->scale);  
**+** }  
**+** }  
**+**}

Actorクラスには描画に必要な情報がひととおり揃っていますので、この関数はそれらをShader::ProgramのBindTexture関数とDraw関数に指定して実行するだけです。

## std::vector<Actor\*>を空にする

最後に、std::vector<Actor\*>に含まれる全てのアクターを終了させ、vectorを空にする関数を定義します。RenderActorList関数の定義の下に、次のプログラムを追加してください。

shader.Draw(meshList.Get(actor->mesh),  
 actor->position, actor->rotation, actor->scale);  
 }  
 }  
}  
**+**  
**+**/\*\*  
**+**\* アクターリストを空にする.  
**+**\*  
**+**\* @param actorList 空にするアクターのリスト.  
**+**\*/  
**+**void ClearActorList(std::vector<Actor\*>& actorList)  
**+**{  
**+** for (auto& actor : actorList) {  
**+** delete actor;  
**+** }  
**+** actorList.clear();  
**+**}

アクターを削除する場合、helthのチェックは不要なのでnullptrかどうかだけを調べています。  
その後はdeleteを呼び出してアクターを削除します。全てのアクターを削除したら、std::vector::clear(えすてぃーでぃー・べくたー・くりあ)関数を実行してvectorの内容を削除します。

**[重要]** 配列やstd::vectorなどにnewやmallocで作成したポインタを保持させている場合、それらを使い終わったときは必ずdeleteやfreeを実行してポインタが指すメモリを開放しなければなりません。これを忘れると、使われないメモリが残ったままになり、徐々にメモリを食いつぶして最後にはプログラムが異常終了する可能性があります。

# アクターの派生クラスを作る

## 仮想関数＝役割によって異なる動作をする関数

ゲーム中にはさまざまな役割を持ったアクターが登場します。それは例えば、プレイヤーや敵、弾などです。これらのアクターは、その役割に応じた方法で状態を更新していかねばなりません。その方法は、プレイヤーならキーボード操作によって移動する、敵ならプレイヤーを追いかける、弾ならまっすぐ飛ぶ、といったものになるでしょう。ActorクラスのUpdate関数にvirtual(ばーちゃる)を指定したのは、役割によって更新関数を切り替えられるようにするためです。

派生クラスのアドレスは、基底クラスのポインタ変数に格納することができます。基底クラスのポインタ変数に対して「->(アロー)」演算子を使うと、そのクラスのメンバ関数を呼び出すことができます。このとき、通常のメンバ関数は、ポインタ変数のクラスで定義されたもの、つまり基底クラスのメンバ関数が実行されます。しかし、メンバ関数にvirtualキーワードが付与されている場合、ポインタ変数が実際に指しているクラス、つまり派生クラスで定義されたメンバ関数が実行されるのです。  
以下のプログラム例では、Base(べーす)クラスからDerived(でらいぶど)クラスを派生させています。  
どちらも2つのメンバ関数を持っていてます。一方はPrintName(ぷりんと・ねーむ)という名前の普通のメンバ関数です。もう片方はPrintRealName(ぷりんと・りある・ねーむ)という名前で、virtualを付けているので仮想メンバ関数だと分かります。

/\* 仮想メンバ関数のプログラム例 \*/  
#include <iostream>  
  
class Base {  
public:  
 void PrintName() { std::cout << “Base\n”; }  
 virtual void PrintRealName() { std::cout << “Base\n”; }  
};  
  
class Derived : public Base {  
public:  
 void PrintName() { std::cout << “Derived\n”; }  
 virtual void PrintRealName() override { std::cout << “Derived\n”; }  
};  
  
int main() {  
 Base\* p = new Derived;  
 p->PrintName(); // 「Base」が表示される  
 p->PrintRealName(); // 「Derived」が表示される  
 delete p;  
}

上記のプログラムを実行すると、最初の行には「Base」と出力されるでしょう。なぜなら、PrintName関数は普通のメンバ関数なので、コンピューターは基本どおり変数pの型に従ってBase::PrintName関数を実行するからです。  
ところが、次の行には「Derived」と出力されるはずです。PrintRealName関数は仮想メンバ関数なので、コンピューターはpの型を見ただけでは実行する関数を決められません。そこで、pが実際に指しているクラスを調べます。すると、Derivedクラスだということが分かるのでDerived::PrintRealName関数を実行するのです。

**［補足］** Derived(でらいぶど)には動詞derive(でらいぶ)の過去形、過去分詞としての「～を得る」、形容詞としての「由来の、派生の」の2つの用法がありますが、クラスを説明する文脈で使われる場合はほぼ後者だと思って間違いないでしょう。

## PlayerActorを作成する

最初にプレイヤーが操作するアクターを作ります。MainGameScene.hに次のプログラムを追加してください。

#ifndef MAINGAMESCENE\_H\_INCLUDED  
 #define MAINGAMESCENE\_H\_INCLUDED  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Shader.h"  
 #include "MeshList.h"  
 #include "Scene.h"  
**+**#include "Actor.h"  
**+**  
**+**/\*\*  
**+**\* プレイヤーが操作するアクター.  
**+**\*/  
**+**class PlayerActor : public Actor  
**+**{  
**+**public:  
**+** virtual ~PlayerActor() = default;  
**+** virtual void Update(float deltaTime) override;  
**+**};  
  
 /\*\*  
 \* メインゲーム画面.  
 \*/  
 class MainGameScene : public Scene

次に、メンバ関数を定義します。MainGameScene.cppに次のプログラムを追加してください。

/\*\*  
 \* @file MainGameScene.cpp  
 \*/  
 #include "MainGameScene.h"  
 #include <glm/gtc/matrix\_transform.hpp>  
**+  
+**  
**+**/\*\*  
**+**\* プレイヤーの状態を更新する.  
**+**\*  
**+**\* @param deltaTime 経過時間.  
**+**\*/  
**+**void PlayerActor::Update(float deltaTime)  
**+**{  
**+** // プレイヤーが死んでいたら、地面に倒れる(-90度になる)まで「倒れ具合」を大きくしていく.  
**+** if (health <= 0) {  
**+** rotation.x -= glm::radians(45.0f) \* deltaTime;  
**+** if (rotation.x < glm::radians(-90.0f)) {  
**+** rotation.x = glm::radians(-90.0f);   
**+** }  
**+** }  
**+**  
**+** Actor::Update(deltaTime);  
**+**}  
  
 /\*\*  
 \* 初期化.  
 \*/  
 bool MainGameScene::Initialize()

まず、プレイヤーが死んでいたら(healthが0以下になっていたら)、downAngle(だうん・あんぐる)という変数を0度から徐々に-90度まで変化させます。これはアクターを前のめりに倒れさせるための準備です。

その後、基底クラスのUpdate関数を実行します。基底クラスのメンバ関数を指定するには、上記のプログラムにあるようにメンバ関数名の手前に「クラス名::」を付けます。この書き方は、常に必要な処理は基底クラスにだけ書いておく、という用途でよく使います。

## MainGameSceneにプレイヤーアクターを追加する

作成したクラスの変数をMainGameSceneに追加しましょう。  
MainGameScene.hに次のプログラムを追加してください。

Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;  
   
 float angleY = 0;  
+  
+ PlayerActor player;  
 };  
   
 #endif // MAINGAMESCENE\_H\_INCLUDED

次にplayer変数を初期化します。  
MainGameScene.cppを開き、MainGameScene::Initialize関数に次のプログラムを追加してください。

lights.spot.dirAndCutOff[0] = glm::vec4(  
 glm::normalize(glm::vec3(-1, -2, -2)), std::cos(glm::radians(20.0f)));  
 lights.spot.color[0] = glm::vec3(0.4f, 0.8f, 1.0f) \* 200.0f;  
+  
+ player.Initialize(4, texHuman.Get(), 10,  
+ glm::vec3(8, 0, 8), glm::vec3(0), glm::vec3(1));  
+ player.colLocal =  
+ { glm::vec3(-0.5f, 0.0f, -0.5f), glm::vec3(1.0f, 1.7f, 1.0f) };  
  
 return true;  
 }

なお、Actor::Initialize関数の最初の引数はメッシュIDです。上記のプログラムでは4としていますが、みなさんのプログラムではメッシュIDが違うかもしれません。その場合はhuman.obj用のメッシュIDを指定してあげてください。

## プレイヤーの状態を更新する

プレイヤーの状態を更新するプログラムを追加しましょう。  
MainGameScene::Updata関数に、次のプログラムを追加してください。

void MainGameScene::Update()  
 {  
+ const float deltaTime = (float)GLFWEW::Window::Instance().DeltaTime();  
+  
 // モデルのY軸回転角を更新.  
 static float angleY = 0;  
 angleY += glm::radians(1.0f);  
 if (angleY > glm::radians(360.0f)) {  
 angleY -= glm::radians(360.0f);  
 }  
+  
+ // プレイヤーの状態を更新.  
+ player.Update(deltaTime);  
 }

GLFWEW::Windowクラスから経過時間を取得し、PlayerActor::Update関数の引数にします。

## プレイヤーを描画する

プレイヤーを描画します。MainGameScene::Render関数に、次のプログラムを追加してください。

progLighting.Use();  
  
 meshList.BindVertexArray();  
  
 // 光源を設定する.  
 progLighting.SetLightList(lights);  
+  
+ progLighting.BindTexture(0, player.texture);  
+ progLighting.Draw(meshList.Get(player.mesh),  
+ player.position, player.rotation, player.scale);  
  
 // ポイント・ライトの位置が分かるように適当なモデルを表示.  
 {  
 progSimple.Use();  
 progSimple.BindTexture(0, texId.Get());

プレイヤーは1体だけなので、Shader::Programを直接使って描画しています。

## プレイヤーアクターを操作する

続いて、プレイヤーアクターを動かすプログラムを追加しましょう。  
MainGameScene::ProcessInput関数を次のように変更してください。

void MainGameScene::ProcessInput()  
 {  
- // 0番のポイント・ライトを移動する.  
- const float speed = 0.05f;  
- if (window.IsKeyPressed(GLFW\_KEY\_A)) {  
- lights.point.position[0].x -= speed;  
- } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
- lights.point.position[0].x += speed;  
- }  
- if (window.IsKeyPressed(GLFW\_KEY\_LEFT\_SHIFT)) {  
- if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
- lights.point.position[0].y += speed;  
- } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
- lights.point.position[0].y -= speed;  
- }  
- } else {  
- if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
- lights.point.position[0].z -= speed;  
- } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
- lights.point.position[0].z += speed;  
- }  
- }  
+ // プレイヤーを移動する.  
+ player.velocity = glm::vec3(0);  
+ if (window.IsKeyPressed(GLFW\_KEY\_A)) {  
+ player.velocity.x = -1;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_D)) {  
+ player.velocity.x += 1;  
+ }  
+ if (window.IsKeyPressed(GLFW\_KEY\_W)) {  
+ player.velocity.z = -1;  
+ } else if (window.IsKeyPressed(GLFW\_KEY\_S)) {  
+ player.velocity.z = 1;  
+ }  
+ if (player.velocity.x || player.velocity.z) {  
+ player.velocity = glm::normalize(player.velocity);  
+   
+ // ショットボタンが押されていなければ方向転換.  
+ if (!window.IsKeyPressed(GLFW\_KEY\_SPACE)) {  
+ player.rotation.y = std::atan2(-player.velocity.z, player.velocity.x);  
+ player.rotation.y -= glm::radians(90.0f);  
+ }  
+ const float speed = 10.0f;  
+ player.velocity \*= speed;  
 }

プレイヤーとライトが同時に動くとややこしいですから、ライトの移動処理は削除します。  
今回は、スペースキーが押され続けている場合はプレイヤーの向きを維持するようにしてみました。こうすることで、弾を撃ちながら後ろに下がるといった行動ができます。

移動ベクトル(velocity)から向きを計算するにはstd::atan2(えすてぃーでぃー・えー・たん・つー)を使います。この関数に、向きをY,Xのベクトルで指定すると、そのベクトルの向きを弧度法(ラジアン)の値で返してくれます。今回はX方向とZ方向から向きを計算したいので、Yの代わりにZを使っています。このとき、Zの符号を逆転させている理由は、OpenGLの座標系ではZ軸の上方向はマイナス方向になっていて、Y軸(上がプラス方向)とは逆だからです。  
なお、human.objモデルでは奥方向が正面ですが、数学では右が0度なので、右を向かせるために、角度の計算後に回転角度から90度引いています。

プログラムが書けたら、**ビルドして実行してください。**プレイヤーアクターが操作できたら成功です。

## 常にプレイヤーが見えるようにする

視点の位置が固定されているので、プレイヤーがちょっと遠くに移動すると画面外に行ってしまいます。  
ゲームとしてこれではまずいので、視点が常にプレイヤーを追いかけるようにしましょう。  
まずはMainGameScene.hに視点の変数を追加します。MainGameSceneクラスの定義に、次のプログラムを追加してください。

Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;  
   
 float angleY = 0;  
+ glm::vec3 viewPos;  
  
 PlayerActor player;  
 };  
   
 #endif // MAINGAMESCENE\_H\_INCLUDED

次に、視点を更新します。  
MainGameScene::Update関数に、次のプログラムを追加してください。

// モデルのY軸回転角を更新.  
 static float angleY = 0;  
 angleY += glm::radians(1.0f);  
 if (angleY > glm::radians(360.0f)) {  
 angleY -= glm::radians(360.0f);  
 }  
   
 // プレイヤーの状態を更新.  
 player.Update(deltaTime);  
+  
+ // 視点の位置を更新.  
+ const glm::vec3 viewOffset(0, 20, 10);  
+ viewPos = player.position + viewOffset;  
 }

最後に、更新した視点を使うようにしましょう。MainGameScene::Render関数を、次のように変更してください。

glClearColor(0.1f, 0.3f, 0.5f, 1.0f);  
 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  
   
- // 視点を設定する.  
- const glm::vec3 viewPos(20, 30, 30);  
-  
 // 座標変換行列を作成する.  
 const glm::mat4x4 matProj =  
 glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 500.0f);  
 const glm::mat4x4 matView =  
- glm::lookAt(viewPos, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));  
+ glm::lookAt(viewPos, player.position, glm::vec3(0, 1, 0));

プログラムが書けたら、**ビルドして実行してください。**プレイヤーアクターを操作したとき、常にアクターを見るように視点が移動していたら成功です。

# 弾のアクターを追加する

## 弾アクタークラスを定義する

ここからは弾のアクターを作成していきますが、その前に弾のモデルデータが必要です。ウェブから適当なファイルを探してくるか、Blenderを使って適当な弾丸モデルを作成し、Resフォルダに追加しておいてください。なお、テキストでは追加した弾丸モデルのファイル名は「bullet.obj」としますが、好きな名前を使ってもらって構いません。

弾丸モデルを追加したら、早速アクターを追加していきましょう。  
MainGameScene.hに次のプログラムを追加してください。

virtual ~PlayerActor() = default;  
 virtual void Update(float deltaTime) override;  
};  
  
**+**/\*\*  
**+**\* プレイヤーから発射される弾のアクター.  
**+**\*/  
**+**class BulletActor : public Actor  
**+**{  
**+**public:  
**+** virtual ~BulletActor() = default;  
**+** virtual void Update(float deltaTime) override;  
**+**};  
**+**  
  
 /\*\*  
 \* メインゲーム画面.  
 \*/  
 class MainGameScene : public Scene

プレイヤーアクターと同じく、弾アクターにもUpdate関数を定義します。  
MainGameScene.cppに次のプログラムを追加してください。

if (rotation.x < glm::radians(-90.0f)) {  
 rotation.x = glm::radians(-90.0f);   
 }  
 }  
  
 Actor::Update(deltaTime);  
}  
  
**+**/\*\*  
**+**\* プレイヤーの弾の状態を更新する.  
**+**\*/  
**+**void BulletActor::Update(float deltaTime)  
**+**{  
**+** const float mapSize = 20;  
**+** for (size\_t i = 0; i < 3; ++i) {  
**+** if (position[i] < -mapSize || position[i] > mapSize) {  
**+** health = 0;  
**+** break;  
**+** }  
**+** }  
**+** Actor::Update(deltaTime);  
**+**}  
**+**  
 /\*\*  
 \* 初期化.  
 \*/  
 bool MainGameScene::Initialize()

マップの広さを20m✕20mとして、弾がその範囲を超えたら消えるようにしています(マップの広さは適宜変更してください)。

## MainGameSceneに弾アクターを追加する

弾は何発も画面に登場する可能性があります。ですから、弾アクターはstd::vectorを使って追加します。まずはvectorヘッダをインクルードしましょう。MainGameScene.hに次のプログラムを追加してください。

#ifndef MAINGAMESCENE\_H\_INCLUDED  
 #define MAINGAMESCENE\_H\_INCLUDED  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Shader.h"  
 #include "MeshList.h"  
 #include "Scene.h"  
#include "Actor.h"  
**+**#include <vector>  
  
/\*\*  
\* プレイヤーが操作するアクター.  
\*/  
class PlayerActor : public Actor

続いて、MainGameSceneクラスの定義に、次のプログラムを追加してください。

Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;  
  
 float angleY = 0;  
  
 PlayerActor player;  
+ std::vector<Actor\*> playerBullets;  
 };  
  
 #endif // MAINGAMESCENE\_H\_INCLUDED

これで弾を作成する準備が整いました。

## 弾アクターを初期化する

playerBulletsが保持する型はActor\*なので、そのままでは使い物になりません。そこで、初期化を行います。MainGameScene::Initialize関数に、次のプログラムを追加してください。

player.Initialize(4, texHuman.Get(), 10,  
 glm::vec3(8, 0, 8), glm::vec3(0), glm::vec3(1));  
  
+ playerBullets.resize(128);  
+ for (auto& bullet : playerBullets) {  
+ bullet = new BulletActor;  
+ }  
+  
 return true;  
 }

vector::resize関数を使って、弾アクターを128個入れられる状態にし、forとnewを使ってそれぞれの要素にBulletActorを作成しています。

## 弾アクターを空にする

newで作成した変数は、不要になったらdeleteで破棄しなければなりません。  
そこで、MainGameScene::Finalize関数に、次のプログラムを追加してください。

/\*\*  
 \* 終了.  
 \*/  
 void MainGameScene::Finalize()  
 {  
+ ClearActorList(playerBullets);  
 }

std::vector<Actor\*>を空にする関数は1.8節で作成済みでしたね。ですから、それを使うだけでO.K.です。

## 弾アクターを更新する

次に弾アクターの更新を作成します。MainGameScene::Update関数に、次のプログラムを追加してください。

// モデルのY軸回転角を更新.  
 static float angleY = 0;  
 angleY += glm::radians(1.0f);  
 if (angleY > glm::radians(360.0f)) {  
 angleY -= glm::radians(360.0f);  
 }  
   
 // プレイヤーの状態を更新.  
 player.Update(deltaTime);  
+  
+ // プレイヤーの弾の状態を更新.  
+ UpdateActorList(playerBullets, deltaTime);  
 }

ここは作ってある関数を呼び出すだけですね。

続いて、弾アクターを表示します。MainGameScene::Render関数に、次のプログラムを追加してください。

progLighting.BindTexture(0, player.texture);  
 progLighting.Draw(meshList.Get(player.mesh),  
 player.position, player.rotation, player.scale);  
  
+ RenderActorList(playerBullets, progLighting, meshList);  
+  
 // ポイント・ライトの位置が分かるように適当なモデルを表示.  
 {  
 progSimple.Use();  
 progSimple.BindTexture(0, texId.Get());

これも作成済みの関数を呼び出すだけです。簡単ですね。

## 弾アクターを発射する

これで弾アクターを動かす準備は整いました。早速発射してみましょう。  
まずは、スペースキーを押しっぱなしで連射できるように、発射間隔タイマーを追加します。  
MainGameScene.hに次のプログラムを追加してください。

Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;  
  
 float angleY = 0;  
  
 PlayerActor player;  
 std::vector<Actor\*> playerBullets;  
+ float playerBulletTimer = 0;  
 };  
  
 #endif // MAINGAMESCENE\_H\_INCLUDED

そして、タイマーの更新処理を追加します。  
MainGameScene::Update関数に、次のプログラムを追加してください。

// プレイヤーの状態を更新.  
 player.Update(deltaTime);  
  
+ // プレイヤーの弾の発射タイマーを更新.  
+ if (playerBulletTimer > 0) {  
+ playerBulletTimer -= deltaTime;  
+ }  
+  
 // 視点の位置を更新.  
 const glm::vec3 viewOffset(0, 20, 10);  
 viewPos = player.position + viewOffset;  
 }

最後に、キーが押されたら弾を発射するプログラムを作成します。MainGameScene::ProcessInput関数に、次のプログラムを追加してください。

// ショットボタンが押されていなければ方向転換.  
 if (!window.IsKeyPressed(GLFW\_KEY\_SPACE)) {  
 player.rotation.y = std::atan2(-player.velocity.z, player.velocity.x)  
 player.rotation.y -= glm::pi<float>() / 2;  
 }  
 const float speed = 10.0f;  
 player.velocity \*= speed;  
 }  
+  
+ // ショットボタンが押されていたら弾を発射.  
+ if (window.IsKeyPressed(GLFW\_KEY\_SPACE)) {  
+ if (playerBulletTimer <= 0) {  
+ Actor\* bullet = FindAvailableActor(playerBullets);  
+ if (bullet) {  
+ const int meshId = 6; // 弾のメッシュID.  
+ const float speed = 40.0f; // 弾の移動速度(m/秒).  
+ const glm::mat4 matRotY =  
+ glm::rotate(glm::mat4(1), player.rotation.y, glm::vec3(0, 1, 0));  
+ bullet->Initialize(meshId, texBullet.Get(), 1,  
+ player.position, player.rotation, glm::vec3(1));  
+ bullet->velocity = matRotY \* glm::vec4(0, 0, -speed, 1);  
+ bullet->colLocal =  
+ { glm::vec3(-0.25f, -0.25f, -0.25f), glm::vec3(1, 1, 1) };  
+  
+ playerBulletTimer = 1.0f / 8.0f; // 秒間8連射.  
+ }  
+ }  
+ } else {  
+ playerBulletTimer = 0.0f;  
+ }  
 }

上記のプログラムでは弾のメッシュIDは6にしていますが、実際のプログラムではみなさん自身が設定した番号を指定してあげてください。  
弾の発射方向はプレイヤーの向きによって異なります。そこで、プレイヤーのY軸回転角度の値から回転行列を作成し、弾の移動方向を回転させています。  
弾を発射したら、playerBulletTimerに次に弾を発射するまでの時間を設定します。なお、ショットボタンを押していなかった場合にplayerBulletTimerに0を設定しているのは、キーが離されて次に押されたとき、即座に弾が発射されるようにするためです。

プログラムが書けたら、**ビルドして実行してください。**  
スペースキーを押すとプレイヤーが向いている方向に弾が発射され、その弾がマップの端で消えたら成功です。

# 敵のアクターを追加する

## ゾンビアクタークラスを定義する

弾を発射できるだけではあまり意味がありません。なんらかの標的が必要ですよね。そこでゾンビの登場です。MainGameScene.hに次のプログラムを追加してください。

virtual ~BulletActor() = default;  
 virtual void Update(float deltaTime) override;  
 };  
  
/\*\*  
\* 敵(ゾンビ)のアクター.  
\*/  
class ZombieActor : public Actor  
{  
public:  
 virtual ~ZombieActor() = default;  
 virtual void Update(float deltaTime) override;  
  
 Actor\* target = nullptr;  
 float attackingTimer = 5.0f;  
 bool isAttacking = false;  
 float baseSpeed = 1.0f;  
};  
  
 /\*\*  
 \* メインゲーム画面.  
 \*/  
 class MainGameScene : public Scene

ゾンビも標的にされるだけでは不愉快でしょうから、プレイヤーを追いかけて攻撃する能力を付けることにしましょう。それを実現するために、いくつかのメンバ変数を付け足しています。

次に、メンバ関数を定義します。MainGameScene.cppに次のプログラムを追加してください。

if (position[i] < -mapSize || position[i] > mapSize) {  
 health = 0;  
 break;  
 }  
 }  
 Actor::Update(deltaTime);  
 }  
  
+/\*\*  
+\* ゾンビの状態を更新する.  
+\*/  
+void ZombieActor::Update(float deltaTime)  
+{  
+ if (!target) {  
+ return;  
+ }  
+  
+ const float moveSpeed = baseSpeed \* 2.0f;  
+ const float rotationSpeed = baseSpeed \* glm::radians(60.0f);  
+ const float frontRange = glm::radians(15.0f);  
+  
+ // ターゲット(プレイヤー)の方向を調べる.  
+ const glm::vec3 v = target->position - position;  
+ const glm::vec3 vTarget = glm::normalize(v);  
+ float radian = std::atan2(-vTarget.z, vTarget.x) - glm::radians(90.0f);  
+ if (radian <= 0) {  
+ radian += glm::radians(360.0f);  
+ }  
+ const glm::vec3 vZombieFront = glm::rotate(  
+ glm::mat4(1), rotation.y, glm::vec3(0, 1, 0)) \* glm::vec4(0, 0, -1, 1);  
+  
+ // ターゲットが正面にいなかったら、正面に捉えるよう左右に旋回.  
+ if (std::abs(radian - rotation.y) > frontRange) {  
+ const glm::vec3 vRotDir = glm::cross(vZombieFront, vTarget);  
+ if (vRotDir.y >= 0) {  
+ rotation.y += rotationSpeed \* deltaTime;  
+ if (rotation.y >= glm::radians(360.0f)) {  
+ rotation.y -= glm::radians(360.0f);  
+ }  
+ } else {  
+ rotation.y -= rotationSpeed \* deltaTime;  
+ if (rotation.y < 0) {  
+ rotation.y += glm::radians(360.0f);  
+ }  
+ }  
+ }  
+  
+ // 十分に接近していなければ移動する. 接近していれば攻撃する.  
+ if (glm::length(v) > 0.5f) {  
+ velocity = vZombieFront \* moveSpeed;  
+ } else {  
+ velocity = glm::vec3(0); // 接近しているのでもう移動しない.  
+ // 定期的に攻撃状態になる.  
+ if (isAttacking) {  
+ isAttacking = false;  
+ attackingTimer = 5.0f; // 次の攻撃は5秒後.  
+ } else {  
+ attackingTimer -= deltaTime;  
+ if (attackingTimer <= 0) {  
+ isAttacking = true;  
+ }  
+ }  
+ }  
+  
+ Actor::Update(deltaTime);  
+}  
  
 /\*\*  
 \* 初期化.  
 \*/  
 bool MainGameScene::Initialize()

ゾンビはプレイヤーを追いかけます。  
まずstd::atan2関数でプレイヤーの方向を調べ、自分がそちらを向いていなければ、正面に捉えるように旋回します。旋回方向を決めるためにglm::cross(じーえるえむ・くろす)という関数を使っています。この関数は「ベクトルの外積」を計算します。詳細は省きますが、、ここでは外積のいくつかの性質のうち、「2つの単位ベクトルAとBがなす角をθ(しーた)とすると、glm::cross(A,B)はsinθ(さいん・しーた)を返す」という性質を利用しています。sinθの性質から、θが0～180度では正の値、180～360度では負の値を取りますから、外積を計算することで、プレイヤーがゾンビの左右どちらにいるかが分かるわけです。  
左右が分かればあとは簡単で、左なら角度をプラスし、右ならマイナスするだけです。

旋回動作の次は、距離が十分に近い(0.5m以内)かどうかを調べます。  
近くにいない場合は前方に移動します。十分に近い場合は停止して、攻撃のカウントダウンを始めます。そして、タイマーが0以下になったら攻撃状態になります。

## MainGameSceneにゾンビアクターを追加する

それでは、ゾンビを追加しましょう。MainGameSceneクラスの定義に、次のプログラムを追加してください。

Shader::Program progSimple;  
 Shader::Program progLighting;  
 Shader::LightList lights;  
  
 float angleY = 0;  
  
 PlayerActor player;  
 std::vector<Actor\*> playerBullets;  
 float playerBulletTimer = 0;  
+  
+ std::vector<Actor\*> enemies;  
 };  
  
 #endif // MAINGAMESCENE\_H\_INCLUDED

これでゾンビを作成する準備が整いました。

## ゾンビアクターを初期化する

弾アクターと同様にゾンビアクターも初期化を行います。MainGameScene::Initialize関数に、次のプログラムを追加してください。

playerBullets.resize(128);  
 for (auto& bullet : playerBullets) {  
 bullet = new BulletActor;  
 }  
+ enemies.resize(128);  
+ for (auto& zombie : enemies) {  
+ zombie = new ZombieActor;  
+ }  
  
 return true;  
 }

vector::resize関数を使って、ゾンビアクターを128個入れられる状態にし、forとnewを使ってそれぞれの要素にZombieActorを作成しています。

## ゾンビアクターを空にする

newで作成した変数は、不要になったらdeleteで破棄しなければならないのでしたね。  
そこで、MainGameScene::Finalize関数に、次のプログラムを追加してください。

/\*\*  
 \* 終了.  
 \*/  
 void MainGameScene::Finalize()  
 {  
 ClearActorList(playerBullets);  
+ ClearActorList(enemies);  
 }

弾アクターと同様に、ClearActorList関数を使います。

## ゾンビアクターを更新する

次にゾンビアクターの更新を作成します。MainGameScene::Update関数に、次のプログラムを追加してください。

// プレイヤーの状態を更新.  
 player.Update(deltaTime);  
  
 // プレイヤーの弾の状態を更新.  
 UpdateActorList(playerBullets, deltaTime);  
+  
+ // ゾンビの状態を更新.  
+ UpdateActorList(enemies, deltaTime);  
 }

## ゾンビアクターを表示する

続いて、ゾンビアクターを表示します。MainGameScene::Render関数に、次のプログラムを追加してください。

progLighting.BindTexture(0, player.texture);  
 progLighting.Draw(meshList.Get(player.mesh),  
 player.position, player.rotation, player.scale);  
  
 RenderActorList(playerBullets, progLighting, meshList);  
+ RenderActorList(enemies, progLighting, meshList);  
  
 // ポイント・ライトの位置が分かるように適当なモデルを表示.  
 {  
 progSimple.Use();  
 progSimple.BindTexture(0, texId.Get());

これも作成済みの関数を呼び出すだけです。

## ゾンビを出現させる

いよいよ、ゾンビを出現させます。まずは出現プログラムに使う変数を用意しますが、その前にrandomヘッダをインクルードしておきます。MainGameScene.hに次のプログラムを追加してください。

#ifndef MAINGAMESCENE\_H\_INCLUDED  
 #define MAINGAMESCENE\_H\_INCLUDED  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Shader.h"  
 #include "MeshList.h"  
 #include "Scene.h"  
#include "Actor.h"  
#include <vector>  
**+**#include <random>  
  
/\*\*  
\* プレイヤーが操作するアクター.  
\*/  
class PlayerActor : public Actor

randomはゾンビをランダムな位置に出現させるために使います。  
それでは変数を追加しましょう。MainGameScene.hに次のプログラムを追加してください。

float angleY = 0;  
  
 PlayerActor player;  
 std::vector<Actor\*> playerBullets;  
 float playerBulletTimer = 0;  
  
 std::vector<Actor\*> enemies;  
+  
+ std::mt19937 random;  
+  
+ int stageNo = 1;  
+ int score = 0;  
+ float enemySpeed = 1.0f; // ゾンビの移動速度.  
+ float enemyPoppingInterval = 15.0f; // ゾンビ出現間隔.  
+ float enemyPoppingTimer = 0.0f; // ゾンビ出現タイマー.  
+ int enemyTotal = 100; // 敵の総数.  
+ int enemyLeft = 100; // 未登場の敵の数. 敵を出現させるたびに減少していく.  
+ int enemyKilled = 0; // 殺した敵の数. この数値がenemyTotalと等しくなったらステージクリア.  
 };  
  
 #endif // MAINGAMESCENE\_H\_INCLUDED

randomは初期化しておく必要があります。MainGameScene::Initialize関数に、次のプログラムを追加してください。

/\*\*  
 \* 初期化.  
 \*/  
 bool MainGameScene::Initialize()  
 {  
+ random.seed(std::random\_device()());  
+  
 MeshList meshList;  
 if (!meshList.Allocate()) {  
 return 1;  
 }

続いて、MainGameScene::Update関数に、次のプログラムを追加してください。

// プレイヤーの弾の状態を更新.  
 UpdateActorList(playerBullets, deltaTime);  
   
 // ゾンビの状態を更新.  
 UpdateActorList(zombies, deltaTime);  
+  
+ // ゾンビの発生.  
+ if (enemyLeft > 0) {  
+ if (enemyPoppingTimer >= 0) {  
+ enemyPoppingTimer -= deltaTime;  
+ } else {  
+ enemyPoppingTimer += enemyPoppingInterval;  
+  
+ int popCount = 10; // 同時に発生するゾンビの最大数.  
+ if (enemyLeft < popCount) {  
+ popCount = enemyLeft;  
+ }  
+ enemyLeft -= popCount;  
+  
+ // マップの前後左右15mの範囲で出現ポイントを決め、そこを中心として前後左右5mの範囲に出現.  
+ std::uniform\_int\_distribution<int> rangeBase(-15, 15);  
+ std::uniform\_int\_distribution<int> range(-5, 5);  
+ glm::vec3 posBase(rangeBase(random), 0, rangeBase(random));  
+ for (int i = 0; i < popCount; ++i) {  
+ ZombieActor\* zombie = (ZombieActor\*)FindAvailableActor(enemies);  
+ if (zombie) {  
+ glm::vec3 pos = posBase + glm::vec3(range(random), 0, range(random));  
+ zombie->Initialize(4, texHuman.Get(), 5,  
+ pos, glm::vec3(0), glm::vec3(1));  
+ zombie->colLocal =  
+ { glm::vec3(-0.5f, 0, -0.5f), glm::vec3(1, 1.8f, 1) };  
+ zombie->target = &player;  
+ zombie->baseSpeed = enemySpeed;  
+ } else {  
+ ++enemyLeft; // 出現できなかった場合、残り数を増やして次回の出現を待つ.  
+ }  
+ }  
+ }  
+ }  
 }

ゾンビは同時に最大10体まで出現するようにしています。そのため、forを使って出現数だけループさせます。なお、出現位置はランダムなので、ゾンビが重なって出現することがあるかもしれません。

プログラムが書けたら、**ビルドして実行してください。**  
ゾンビ(といってもプレイヤーと見分けが付きませんが)が出現し、プレイヤーを追いかけてきたら成功です。

**[課題01]** ゾンビ用のテクスチャを作成し、ゾンビだと分かるようにしてください。

# 衝突判定

## 衝突判定関数を作成する

弾がゾンビを撃ち抜けるようにするには、弾とゾンビの衝突判定を行わなければなりません。  
2Dゲームでは2次元の矩形同士で衝突判定をしていましたが、今回は3Dなので、直方体同士の衝突判定を作成していきます。

今回は新しくstd::function(えすてぃーでぃー・・ふぁんくしょん)というクラスを使おうと思います。このクラスを使うにはfunctional(ふぁんくしょなる)ヘッダをインクルードします。  
Actor.hに次のプログラムを追加してください。

/\*\*  
 \* @file Actor.h  
 \*/  
 #ifndef ACTOR\_H\_INCLUDED  
 #define ACTOR\_H\_INCLUDED  
 #include <GL/glew.h>  
 #include "Shader.h"  
 #include "MeshList.h"  
 #include <glm/vec3.hpp>  
 #include <vector>  
**+**#include <functional>

続いて関数を定義しましょう。Actor.hに次のプログラムを追加してください。

Actor\* FindAvailableActor(std::vector<Actor\*>&);  
void UpdateActorList(std::vector<Actor\*>&, float);  
void RenderActorList(std::vector<Actor\*>&, Shader::Program&, MeshList&);  
void ClearActorList(std::vector<Actor\*>&);  
**+  
+**using CollisionHandlerType = std::function<void(Actor&,Actor&)>; **+**bool DetectCollision(const Actor&, const Actor&);  
**+**void DetectCollision(  
**+** std::vector<Actor\*>&, std::vector<Actor\*>&, CollisionHandlerType);  
#endif // ACTOR\_H\_INCLUDED

CollisionHandlerType(こりじょん・はんどらー・たいぷ)型を定義するときにstd::functionを使っています。「void(\*)(Actor&, Actor&)」と書く場合との違いは「void(Actor&,Actor&)として呼び出せるのであれば、厳密にvoid(Actor&,Actor&)型の関数でなくても指定できる」ことです。詳しくは実際に使うときにまた説明します。

そして関数を定義します。Actor.cppの末尾に次のプログラムを追加してください。

actor->Finalize();  
 delete actor;  
 }  
 }  
 actorList.clear();  
 }  
+  
+/\*\*  
+\* 2つの直方体の衝突状態を調べる.  
+\*  
+\* @param lhs直方体その1.  
+\* @param rhs直方体その2.  
+\*  
+\* @retval true 衝突している.  
+\* @retval false 衝突していない.  
+\*/  
+bool DetectCollision(const Actor& lhs, const Actor& rhs)  
+{  
+ return  
+ lhs.colWorld.origin.x < rhs.colWorld.origin.x + rhs.colWorld.size.x &&  
+ lhs.colWorld.origin.x + lhs.colWorld.size.x > rhs.colWorld.origin.x &&  
+ lhs.colWorld.origin.y < rhs.colWorld.origin.y + rhs.colWorld.size.y &&  
+ lhs.colWorld.origin.y + lhs.colWorld.size.y > rhs.colWorld.origin.y &&  
+ lhs.colWorld.origin.z < rhs.colWorld.origin.z + rhs.colWorld.size.z &&  
+ lhs.colWorld.origin.z + lhs.colWorld.size.z > rhs.colWorld.origin.z;  
+}

2Dから3DになったことでZ軸の判定が増えていますが、基本的には2D長方形の衝突判定と同じです。  
続いてstd::vector<Actor\*>同士の判定関数を定義します。上記の関数の下に、次のプログラムを追加してください。

lhs.colWorld.origin.z < rhs.colWorld.origin.z + rhs.colWorld.size.z &&  
 lhs.colWorld.origin.z + lhs.colWorld.size.z > rhs.colWorld.origin.z;  
 }  
+  
+/\*\*  
+\* 2つのグループ間で衝突判定を行う.  
+\*  
+\* @param va グループA.  
+\* @param vb グループB.  
+\* @param func A-B間の衝突を処理する関数.   
+\*/  
+void DetectCollision(std::vector<Actor\*>& va, std::vector<Actor\*>& vb,  
+ CollsionHandlerType func)  
+{  
+ for (auto& a : va) {  
+ if (a->health <= 0) {  
+ continue;  
+ }  
+ for (auto& b : vb) {  
+ if (b->health <= 0) {  
+ continue;  
+ }  
+ if (DetectCollision(\*a, \*b)) {  
+ func(\*a, \*b);  
+ if (a->health <= 0) {  
+ break;  
+ }  
+ }  
+ }  
+ }  
+}

## 弾とゾンビの衝突を処理する

関数を用意できたので、実際に衝突判定を行います。  
MainGameScene::Update関数の末尾に、次のプログラムを追加してください。

zombie->target = &player;  
 zombie->baseSpeed = enemySpeed;  
 } else {  
 ++enemyLeft; // 出現できなかった場合、残り数を増やして次回の出現を待つ.  
 }  
 }  
 }  
 }  
+  
+ // プレイヤーの弾と敵の衝突判定.  
+ DetectCollision(playerBullets, enemies, [&](Actor& bullet, Actor& zombie) {  
+ zombie.health -= bullet.health;  
+ bullet.health = 0;  
+ if (zombie.health <= 0) {  
+ score += 200;  
+ ++enemyKilled;  
+ } else {  
+ score += 10;  
+ }  
+ });  
 }

ここで使っている「[&](Actor&, Actor&){ ～ }」の部分は「無名関数」あるいは「ラムダ式」と呼ばれるC++の機能です。ラムダ式を使うと、いちいち関数を定義しなくても、必要な場所に直接関数の内容を書くことができます。ラムダ式は関数の一種なので、基本的には関数の外にある変数を操作することはできません。しかし、先頭の[]の中に「&」を加えることで、ラムダ式が書かれた位置で使える変数や関数ならなんでも使えるようにすることができます。上記のプログラムでは、scoreやenemyKilledという変数がそうです。これらはMainGameSceneのメンバ変数なので通常は触れられません。しかし「&」を付与することでクラスのポインタであるthisがラムダ式に取り込まれ、これらの変数を操作することができるようになります。

プログラムが書けたら、**ビルドして実行してください。**  
スペースキーで弾を発射して、ゾンビを倒すことができれば成功です。ゾンビはちょっと硬いので、数発打ち込む必要があるでしょう。

## ラムダ式とstd::function型

ところで、上記のラムダ式はCollisionHandlerType型の引数として使われていますよね。この型はstd::function型を使って定義されているのでした。実は「&」を付与したラムダ式は、普通の関数ではなくなってしまいます。なぜなら、普通の関数では不可能な、thisなどを取り込む機能を有しているからです。そこでstd::functionの登場です。この型を使うと、ラムダ式が作り出す特殊な関数であっても、引数として使えるようにできるのです。

## ゾンビの攻撃を処理する

次はゾンビの攻撃を判定しましょう。弾とゾンビの衝突判定の下に、次のプログラムを追加してください。

score += 200;  
 ++enemyKilled;  
 } else {  
 score += 10;  
 }  
 });  
+  
+ // ゾンビの攻撃.  
+ for (auto& actor : enemies) {  
+ if (actor->health <= 0) {  
+ continue;  
+ }  
+  
+ // ターゲットの体力が0以下なら攻撃しない.  
+ ZombieActor\* zombie = (ZombieActor\*)actor;  
+ if (zombie->target->health <= 0) {  
+ continue;  
+ }  
+  
+ if (zombie->isAttacking) {  
+ // ゾンビの正面左右45度の範囲にターゲットがいたら、ターゲットの体力を1減らす.  
+ const glm::vec3 vFront =  
+ glm::rotate(glm::mat4(1), zombie->rotation.y, glm::vec3(0, 1, 0)) \*  
+ glm::vec4(0, 0, -1, 1);  
+ const glm::vec3 vTarget = zombie->target->position - zombie->position;  
+ const float angle = std::acos(glm::dot(vFront,glm::normalize(vTarget)));  
+ if (std::abs(angle) < glm::radians(45.0f) &&  
+ glm::length(vTarget) < 1.5f) {  
+ --zombie->target->health;  
+ }  
+ }  
+ }  
 }

ゾンビが攻撃状態にあるとき(isAttackingがtrue)、ゾンビは自分の正面方向の左右45度の範囲に攻撃を仕掛けます。正面方向は、glm::rotate関数で回転行列を作り、それに正面向きの4Dベクトル(0, 0, -1, 1)を掛け合わせることで得られます。ターゲットのいる方向は、ターゲットの座標からゾンビの座標を引いて正規化することで得られます。  
そして、この2つのベクトル間の角度を得るためにglm::dot(じーえるえむ・どっと)関数とstd::acos(えすてぃーでぃー・・えー・こす)関数を使っています。glm::dotは内積を計算します。単位ベクトル同士の内積はcosθになります(外積はsinθになるのでした)。std::acosはcosθからθを計算する関数なので、これらを組み合わせれば角度が得られるわけです。

ターゲットの角度が分かったら、その角度が攻撃範囲内かどうかを調べます。そして、範囲内であれば、ターゲットの体力を減らします。

プログラムが書けたら、**ビルドして実行してください。**  
ゾンビに囲まれた状態でしばらく放置して、プレイヤーが倒れたら成功です。

## Shader::Program::Draw関数の修正

プレイヤーがやられたとき、倒れ方がおかしい場合があることに気がついたでしょうか。  
おかしくなるのは、回転行列を掛ける順番に問題があるからです。

行列の掛け算では、掛ける順番を間違えると思ったとおりの動きになりません。これは回転行列も例外ではありません。例えばアクターが倒れるときは、現在向いている方向を基準に倒れてほしいです。しかし、そうならない場合があります。というのは、現在Shader::Program::Draw関数ではZ→Y→Xの順番で回転を行っていますが、このうち「Y→X」の順番に問題があるんです。  
まず、Y軸回転は現在の向きを示します。例えば左を向いたとします。次に倒れる角度なのですが、ここから左向きに倒すにはZ軸回転をさせます。しかし、アクターが奥を向いている場合、奥向きに倒すにはX軸回転をさせなければなりません。さらに、斜めを向いている場合はX,Y,Zの全ての軸に適切な角度を指定する必要があります。これでは計算が大変です。

そこで、回転の順番を「X→Z→Y」に変更しようと思います。  
Shader.cppを開き、Shader::Program::Draw関数を次のように変更してください。

void Program::Draw(const Mesh& mesh,  
 const glm::vec3& t, const glm::vec3& r, const glm::vec3& s)  
 {  
 if (id == 0) {  
 return;  
 }  
  
 // モデル行列を計算する.  
 const glm::mat4 matScale = glm::scale(glm::mat4(1), s);  
- const glm::mat4 matRotateX =  
- glm::rotate(glm::mat4(1), r.x, glm::vec3(1, 0, 0));  
- const glm::mat4 matRotateXY =  
- glm::rotate(matRotateX, r.y, glm::vec3(0, 1, 0));  
- const glm::mat4 matRotateXYZ =  
- glm::rotate(matRotateXY, r.z, glm::vec3(0, 0, -1));  
+ const glm::mat4 matRotateY =  
+ glm::rotate(glm::mat4(1), r.y, glm::vec3(0, 1, 0));  
+ const glm::mat4 matRotateZY =  
+ glm::rotate(matRotateY, r.z, glm::vec3(0, 0, -1));  
+ const glm::mat4 matRotateXZY =  
+ glm::rotate(matRotateZY, r.x, glm::vec3(1, 0, 0));  
 const glm::mat4 matTranslate = glm::translate(glm::mat4(1), t);  
- const glm::mat4 matModel = matTranslate \* matRotateXYZ \* matScale;  
+ const glm::mat4 matModel = matTranslate \* matRotateXZY \* matScale;  
  
 // モデル・ビュー・プロジェクション行列を計算し、GPUメモリに転送する.  
 const glm::mat4 matMVP = matVP \* matModel;  
 glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
   
 // モデル座標系における指向性ライトの方向を計算し、GPUメモリに転送する.  
 if (locDirLightDir >= 0) {  
- const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXYZ));  
+ const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXZY));  
 const glm::vec3 dirLightDirOnModel =  
 matInvRotate \* lights.directional.direction;  
 glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
 }  
  
 // モデル座標系におけるポイントライトの座標を計算し、GPUメモリに転送する.  
 if (locPointLightPos >= 0) {  
 const glm::mat4 matInvModel = glm::inverse(matModel);  
 glm::vec3 pointLightPosOnModel[8];  
 for (int i = 0; i < 8; ++i) {  
 pointLightPosOnModel[i] =  
 matInvModel \* glm::vec4(lights.point.position[i], 1);  
 }  
 glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
 }  
  
 // モデル座標系におけるスポットライトの座標を計算し、GPUメモリに転送する.  
 if (locSpotLightPos >= 0 && locSpotLightDir >= 0) {  
- const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXYZ));  
+ const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXZY));  
 const glm::mat4 matInvModel = glm::inverse(matModel);  
 glm::vec4 spotLightPosOnModel[4];  
 glm::vec4 spotLightDirOnModel[4];

プログラムが書けたら、**ビルドして実行してください。**  
プレイヤーが常に前のめりに倒れるようになっていれば成功です(プレイヤーのhealthを1にして試すといいでしょう)。

## ライトの座標変換がスケールの影響を受けないようにする

前回のテキストでライトの座標変換行列の作成方法にミスが有りました。  
これを修正するため、Draw関数を次のように変更してください。

const glm::mat4 matTranslate = glm::translate(glm::mat4(1), t);  
- const glm::mat4 matModel = matTranslate \* matRotateXZY \* matScale;  
+ const glm::mat4 matModelTR = matTranslate \* matRotateXZY;  
+ const glm::mat4 matModel = matModelTR \* matScale;  
  
 // モデル・ビュー・プロジェクション行列を計算し、GPUメモリに転送する.  
 const glm::mat4 matMVP = matVP \* matModel;  
 glUniformMatrix4fv(locMatMVP, 1, GL\_FALSE, &matMVP[0][0]);  
   
 // モデル座標系における指向性ライトの方向を計算し、GPUメモリに転送する.  
 if (locDirLightDir >= 0) {  
 const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXZY));  
 const glm::vec3 dirLightDirOnModel =  
 matInvRotate \* lights.directional.direction;  
 glUniform3fv(locDirLightDir, 1, &dirLightDirOnModel.x);  
 }  
  
 // モデル座標系におけるポイントライトの座標を計算し、GPUメモリに転送する.  
 if (locPointLightPos >= 0) {  
- const glm::mat4 matInvModel = glm::inverse(matModel);  
+ const glm::mat4 matInvModel = glm::inverse(matModelTR);  
 glm::vec3 pointLightPosOnModel[8];  
 for (int i = 0; i < 8; ++i) {  
 pointLightPosOnModel[i] =  
 matInvModel \* glm::vec4(lights.point.position[i], 1);  
 }  
 glUniform3fv(locPointLightPos, 8, &pointLightPosOnModel[0].x);  
 }  
  
 // モデル座標系におけるスポットライトの座標を計算し、GPUメモリに転送する.  
 if (locSpotLightPos >= 0 && locSpotLightDir >= 0) {  
 const glm::mat3 matInvRotate = glm::inverse(glm::mat3(matRotateXZY));  
- const glm::mat4 matInvModel = glm::inverse(matModel);  
+ const glm::mat4 matInvModel = glm::inverse(matModelTR);  
 glm::vec4 spotLightPosOnModel[4];  
 glm::vec4 spotLightDirOnModel[4];  
 for (int i = 0; i < 4; ++i) {  
 const glm::vec3 invDir = matInvRotate \* lights.spot.dirAndCutOff[i];  
 spotLightDirOnModel[i] =  
 glm::vec4(invDir, lights.spot.dirAndCutOff[i].w);  
 const glm::vec3 pos = lights.spot.posAndInnerCutOff[i];  
 spotLightPosOnModel[i] = matInvModel \* glm::vec4(pos, 1);  
 spotLightPosOnModel[i].w = lights.spot.posAndInnerCutOff[i].w;  
 }  
 glUniform4fv(locSpotLightPos, 4, &spotLightPosOnModel[0].x);  
 glUniform4fv(locSpotLightDir, 4, &spotLightDirOnModel[0].x);  
 }

ライトの座標変換にスケールを加えると、スケールが1以外のときにライトの明るさが不自然になってしまいます。そこで、スケール行列を除外しました。

## スケール行列をシェーダーに送る